TCP 粘包是怎么回事，如何处理? UDP 有粘包吗?

有关 TCP 和 UDP 粘包 消息保护边界

在 socket 网络程序中，TCP 和 UDP 分别是面向连接和非面向连接的。

TCP 的 socket 编程，收发两端（客户端和服务器端）都要有一一成对的 socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle 算法），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。

对于 UDP，不会使用块的合并优化算法，这样，实际上目前认为，是由于 UDP 支持的是一对多的模式，所以接收端的 skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的 UDP 包，在每个 UDP 包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。

保护消息边界和流 ，那么什么是保护消息边界和流呢？ 保护消息边界，就是指传输协议把数据当作一条独立的消息在网上 传输,接收端只能接收独立的消息.也就是说存在保护消息边界,接收 端一次只能接收发送端发出的一个数据包. 而面向流则是指无保护消息保护边界的,如果发送端连续发送数据， 接收端有可能在一次接收动作中,会接收两个或者更多的数据包. 我们举个例子来说,例如,我们连续发送三个数据包,大小分别是 2k， 4k , 8k,这三个数据包,都已经到达了接收端的网络堆栈中,如果使 用 UDP 协议,不管我们使用多大的接收缓冲区去接收数据,我们必须有 三次接收动作,才能够把所有的数据包接收完.而使用

TCP 协议,我们 只要把接收的缓冲区大小设置在 14k 以上,我们就能够一次把所有的 数据包接收下来.只需要有一次接收动作. 这就是因为 UDP 协议的保护消息边界使得每一个消息都是独立的.而 流传输,却把数据当作一串数据流,他不认为数据是一个一个的消息. 所以有很多人在使用 tcp 协议通讯的时候,并不清楚 tcp 是基于流的 传输,当连续发送数据的时候,他们时常会认识 tcp 会丢包.其实不然, 因为当他们使用的缓冲区足够大时,他们有可能会一次接收到两个甚 至更多的数据包,而很多人往往会忽视这一点,只解析检查了第一个 数据包,而已经接收的其他数据包却被忽略了.所以大家如果要作这 类的网络编程的时候,必须要注意这一点. 结论： 根据以上所说，可以这样理解，TCP 为了保证可靠传输，尽量减少额外 开销（每次发包都要验证），因此采用了流式传输，面向流的传输， 相对于面向消息的传输，可以减少发送包的数量。从而减少了额外开 销。但是，对于数据传输频繁的程序来讲，使用 TCP 可能会容易粘包。 当然，对接收端的程序来讲，如果机器负荷很重，也会在接收缓冲里 粘包。这样，就需要接收端额外拆包，增加了工作量。因此，这个特 别适合的是数据要求可靠传输，但是不需要太频繁传输的场合（ 两次操作间隔 100ms，具体是由 TCP 等待发送间隔决定的，取决于内核 中的 socket 的写法） 而 UDP，由于面向的是消息传输，它把所有接收到的消息都挂接到缓冲 区的接受队列中，因此，它对于数据的提取分离就更加方便，但是， 它没有粘包机制，因此，当发送数据量较小的时候，就会发生数据包 有效载荷较小的情况，也会增加多次发送的系统发送开销（系统调用， 写硬件等）和接收开销。因此，应该最好设置一个比较合适的数据包 的包长，来进行 UDP 数据的发送。（UDP

最大载荷为 1472，因此最好能 每次传输接近这个数的数据量，这特别适合于视频，音频等大块数据 的发送，同时，通过减少握手来保证流媒体的实时性

UDP 不存在粘包问题，是由于 UDP 发送的时候，没有经过 Negal 算法优化，不会将多个小包合并一次发送出去。另外，在 UDP 协议的接收端，采用了链式结构来记录每一个到达的 UDP 包，这样接收端应用程序一次 recv 只能从 socket 接收缓冲区中读出一个数据包。也就是说，发送端 send 了几次，接收端必须 recv 几次（无论 recv 时指定了多大的缓冲区）。

网络中出现 TCP、UDP 粘包、分包的两点解决办法

粘包产生原因：

先说 TCP：由于 TCP 协议本身的机制（面向连接的可靠地协议-三次握手机制）客户端与服务器会维持一个连接（Channel），数据在连接不断开的情况下，可以持续不断地将多个数据包发往服务器，但是如果发送的网络数据包太小，那么他本身会启用 Nagle 算法（可配置是否启用）对较小的数据包进行合并（基于此，TCP 的网络延迟要 UDP 的高些）然后再发送（超时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就无法区分哪些数据包是客户端自己分开发送的，这样产生了粘包；服务器在接收到数据库后，放到缓冲区中，如果消息没有被及时从缓存区取走，下次在取数据的时候可能就会出现一次取出多个数据包的情况，造成粘包现象（确切来讲，对于基于 TCP 协议的应用，不应用包来描述，而应 用 流来描述），个人认为服务器接收端产生的粘包应该与 linux 内核处理 socket 的方式 select 轮询机制的线性扫描频度无关。

再说 UDP：本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有 Nagle 算法之说了），**他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP 头+IP 头等等发一次数据封装一次）也就没有粘包一说了。**

分包产生的原因就简单的多：可能是 IP 分片传输导致的，也可能是传输过程中丢失部分包导致出现的半包，还有可能就是一个包可能被分成了两次传输，在取数据的时候，先取到了一部分（还可能与接收的缓冲区大小有关系），总之就是一个数据包被分成了多次接收。

解决办法：
粘包与分包的处理方法：

我根据现有的一些开源资料做了如下总结（常用的解决方案）：
一个是采用分隔符的方式，即我们在封装要传输的数据包的时候，采用固定的符号作为结尾符（数据中不能含结尾符），这样我们接收到数据后，如果出现结尾标识，即人为的将粘包分开，如果一个包中没有出现结尾符，认为出现了分包，则等待下个包中出现后 组合成一个完整的数据包，这种方式适合于文本传输的数据，如采用/r/n 之类的分隔符；

另一种是采用在数据包中添加长度的方式，即在数据包中的固定位置封装数据包的长度信息（或可计算数据包总长度的信息），服务器接收到数据后，先是解析包长度，然后根据包长度截取数据包（此种方式常出现于自定义协议中），但是有个小问题就是如果客户端第一个数据包数据长度封装的有错误，那么很可能就会导致后面接收到的所有数据包都解析出错（由于 TCP 建立连接后流式传输机制），只有客户端关闭连接后重新打开才可以消除此问题，我在处理这个问题的时候对数据长度做了校验，会适时的对接收到的有问题的包进行人为的丢弃处理（客户端有自动重发机制，故而在应用层不会导致数据的不完整性）；

另一种不建议的方式是 TCP 采用短连接处理粘包（这个得根据需要来，所以不建议）；

TCP 粘包处理-RingBuf 方法

TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。粘包可能由发送方造成，也可能由接收方造成。TCP 为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据，造成多个数据包的粘连。如果接收进程不及时接收数据，已收到的数据就放在系统接收缓冲区，用户进程读取数据时就可能同时读到多个数据包。因为系统传输的数据是带结构的数据，需要做分包处理。

为了适应高速复杂网络条件，我们设计实现了粘包处理模块，由接收方通过预处理过程，对接收到的数据包进行预处理，将粘连的包分开。为了方便粘包处理，提高处理效率，在接收环节使用了环形缓冲区来存储接收到的数据。其结构如表 1 所示。

表 1 环形缓冲结构

字段名	类型	含义
CS	CRITICAL_SECTION	保护环形缓冲的临界区
pRingBuf	UINT8*	缓冲区起始位置
pRead	UINT8*	当前未处理数据的起始位置
pWrite	UINT8*	当前未处理数据的结束位置
pLastWrite	UINT8*	当前缓冲区的结束位置

环形缓冲跟每个 TCP 套接字绑定。在每个 TCP 的 SOCKET_OBJ 创建时，同时创建一个 PRINGBUFFER 结构并初始化。这时候，pRingBuf 指向环形缓冲区的内存首地址，

pRead、pWrite 指针也指向它。pLastWrite 指针在这时候没有实际意义。初始化之后的结构如图 1 所示。

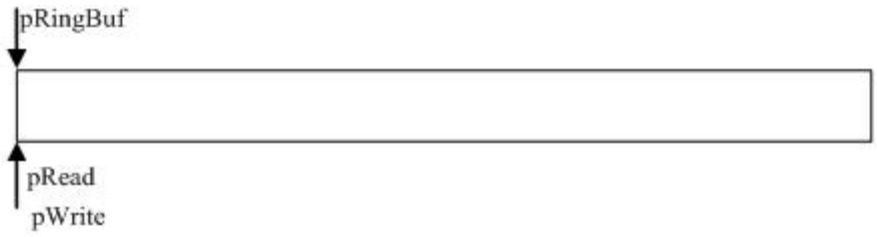

图 1 初始化后的环形缓冲区

在每次投递一个 TCP 的接收操作时，从 RINGBUFFER 获取内存作接收缓冲区，一般规定一个最大值 L1 作为可以写入的最大数据量。这时把 pWrite 的值赋给 BUFFER_OBJ 的 buf 字段，把 L1 赋给 bufLen 字段。这样每次接收到的数据就从 pWrite 开始写入缓冲区，最多写入 L1 字节，如图 2。

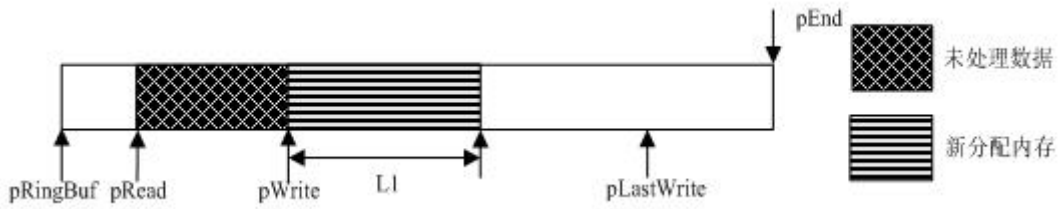

图 2 分配缓冲后的环形缓冲

如果某次分配过程中，pWrite 到缓冲区结束的位置 pEnd 长度不够最小分配长度 L1，为了提高接收效率，直接废弃最后一段内存，标记 pLastWrite 为 pWrite。然后从 pRingBuf 开始分配内存，如图 3。

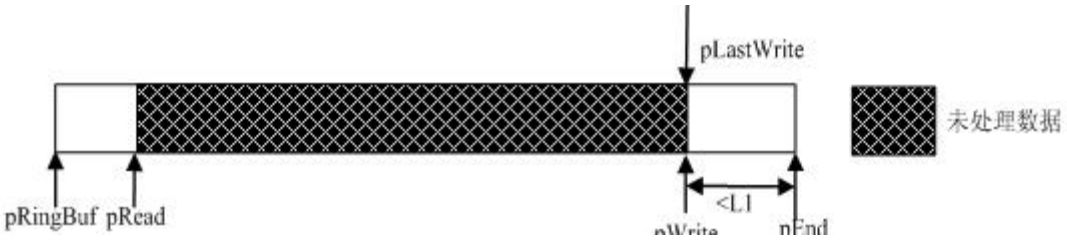

图 3 使用到结尾的环形缓冲

特殊情况下，如果处理包速度太慢，或者接收太快，可能导致未处理包占用大部分缓冲区，没有足够的缓冲区分配给新的接收操作，如图 4。这时候直接报告错误即可。

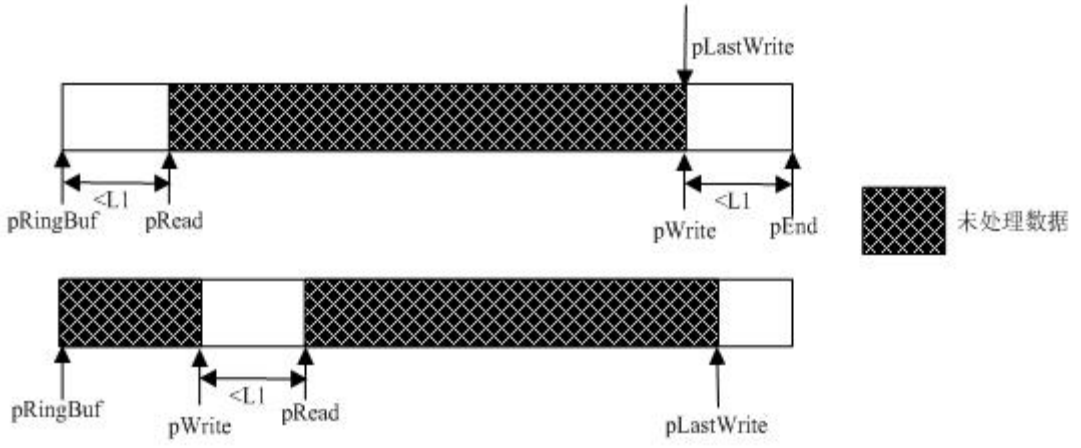

图 4 没有足够接收缓冲的环形缓冲

当收到一个长度为 L 数据包时，需要修改缓冲区的指针。这时候已经写入数据的位置变为（pWrite+L），如图 5。

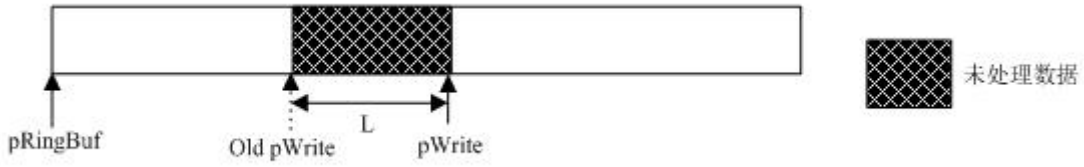

图 5 收到长度为 L 的数据的环形缓冲

分析上述环形缓冲的使用过程，收到数据后的情况可以简单归纳为两种：pWrite>pRead，接收但未处理的数据位于 pRead 到 pWrite 之间的缓冲区；pWrite<pRead，这时候，数据位于 pRead 到 pLastWrite 和 pRingbuf 到 pWrite 之间。这两种情况分别对应图 6、图 7。

首先分析图 6。此时，pRead 是一个包的起始位置，如果 L1 足够一个包头长度，就获取该包的长度信息，记为 L。假如 L1>L，就说明一个数据包接收完成，根据包类型处理包，然后修改 pRead 指针，指向下一个包的起始位置（pRead+L）。这时候仍然类似于之前的状态，于是解包继续，直到 L1 不足一个包的长度，或者不足包头长度。这时退出解包过程，等待后续的数据到来。

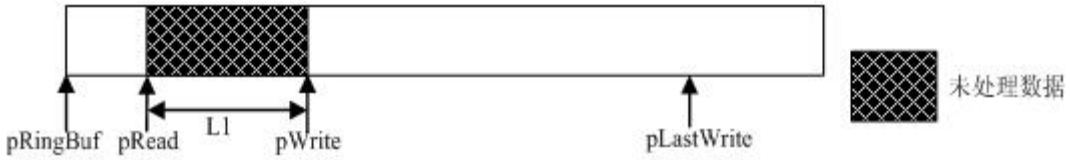

图 6 有未处理数据的环形缓冲（1）

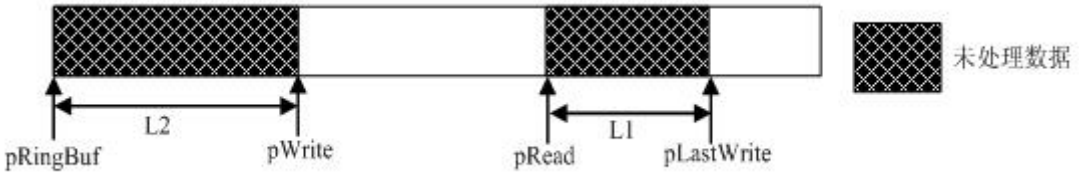

图 7 有未处理数据的环形缓冲（2）

图 8 稍微复杂。首先按照上述过程处理 L1 部分。存在一种情况，经过若干个包处理之后，L1 不足一个包，或者不足一个包头。如果这时（L1+L2）足够一个包的长度，就需要继续处理。另外申请一个最大包长度的内存区 pTemp，把 L1 部分和 L2 的一部分复制到 pTemp，然后执行解包过程。

经过上述解包之后，pRead 就转向 pRingBuf 到 pWrite 之间的某个位置，从而回归情况图 6，继续按照图 6 部分执行解包。